Introducción

José María Alonso Cebrián
Jordi Gay Sensat
Antonio Guzmán Sacristán
Pedro Laguna Durán
Alejandro Martín Bailón
Jordi Serra Ruiz

PID_00208392

Universitat Oberta de Catalunya

Índice

Objetivos

Al finalizar la lectura del presente material, los estudiantes habrán alcanzado las siguientes competencias:

1. Conocer el funcionamiento de la programación a bajo nivel.

2. Conocer la función de los registros del sistema.

3. Comprender cómo funciona un *exploit*.

4. Saber realizar un *exploit* sobre un sistema real.

5. Conocer las técnicas de defensa contra los *exploits* de programación.

6. Saber utilizar las herramientas existentes para el seguimiento de software.

7. Saber cambiar el comportamiento de software en ejecución.

1. Introducción

Las máquinas no se equivocan, nos equivocamos nosotros. No es una visión tenebrista y negativa del ser humano, sino que, como ya dijo Santo Tomás de Aquino hace mucho tiempo, "nada imperfecto puede generar algo perfecto". Creais o no en las tesis de Santo Tomás, lo cierto es que los sistemas operativos, las aplicaciones y, en definitiva, el software en general tienen fallos.

Estos fallos en el software se pueden producir por un mal diseño de la arquitectura, una mala comprobación de las premisas de entorno para las que fue creado o simplemente por una implementación errónea de las especificaciones del mismo.

Un software que funcione será aquel que haga exactamente todo aquello para lo que fue creado y diseñado. Sin embargo, el programa puede ser funcional pero inseguro. Un software seguro será aquel que hace aquello para lo que fue creado y nada más.

Ese algo más que consigue realizar lo vuelve vulnerable y lo convierte en un riesgo para los pilares de la seguridad de aquellas máquinas y/o entornos donde esté corriendo ese software.

Supongamos un programa que recibe dos números como parámetros de entrada para multiplicarlos. Un programa funcional podría ser aquel que recoja los dos números, los almacene en sendas variables, invoque a la función multiplicar y devuelva el resultado en una tercera variable. Esta puede ser la forma en la que muchos han aprendido a programar, pero cuando hablamos de software seguro, esta forma no es correcta. Un programa seguro sería aquel que comprueba exactamente el tamaño de los datos de entrada, comprueba que son dos números, se asegura que son dos números del tipo de datos de las variables que se han reservado y que ambos no desbordan el tipo de datos de la variable que almacenará el resultado.

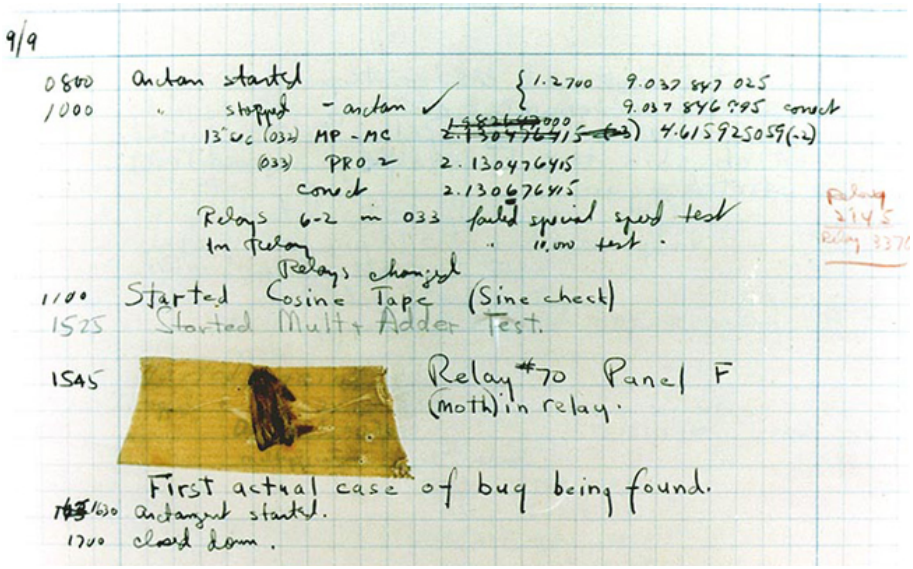
Los límites y premisas que deben comprobarse antes de crear un software son muchos y si esto no se cumple, entonces tendremos un software inseguro.

Durante este curso vamos a repasar los fallos más comunes y cómo se pueden explotar para obtener "esa funcionalidad más" que permita a un atacante sacar provecho de la vulnerabilidad.

1.1. *Bug*

Si retrocedemos hasta los inicios de la computación como ciencia, encontramos ya los primeros problemas con los programas. Aunque ya se habían encontrado fallos en los programas anteriormente, fue en el año 1947, investigando por qué un programa de los primeros ordenadores, el *Mark II*, funcionaba de forma errónea, cuando descubrieron que el mal funcionamiento se debía a una polilla que había en uno de los relés electromagnéticos.

Grace Murray, programadora del *Mark II*, recogió y pegó el insecto con cinta adhesiva en un papel y se refirió a él como el *bug* (bicho en inglés) que causaba el problema. Éste fue el primer caso en que el término *bug* se refería explícitamente a errores en un programa.



Se conoce como *bug*, por tanto, a todos y cada uno de los errores de programación que se detectan en un software.

No todos los *bugs* son iguales, ni mucho menos, y por eso hay que catalogarlos en función de muchos factores. Para conocer correctamente todas las implicaciones de un *bug* se generan las bases de datos de expedientes de seguridad donde los *bugs* van a ser estudiados en profundidad por la comunidad técnica/seguridad.

De igual forma, no todas las bases de datos de expedientes de seguridad son iguales, ni tienen todos los *bugs*, ni todos los *bugs* acaban en bases de datos de expedientes de seguridad. Siempre habrá personas que encuentren vulnerabilidades (o *bugs*) y no las reporten a la comunidad, sino que lo harán a las mafias que llegan a pagar grandes cantidades de dinero por ellas.

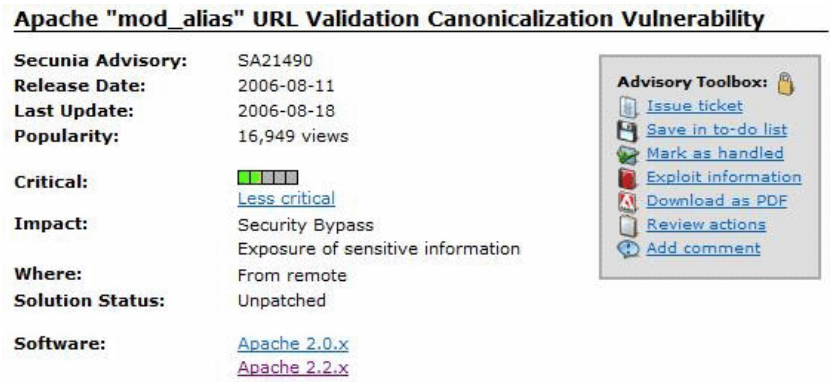
El volumen de software y la rapidez de las versiones de cada programa hacen que documentar todos los *bugs* de todas versiones de todos los programas sea una tarea inabordable. En otras situaciones, el *bug* se produce en código y por motivos de protección de la ventaja competitiva, la propia compañía desarrolladora del software limita, en la medida que puede, la información del mismo.

Otros *bugs* simplemente se encuentran en software privado que sólo una empresa utiliza porque ha sido creado expresamente para ella, es decir, es un software a medida, y salvo que la empresa mantenga una base de datos privada de expedientes de seguridad, estos no se almacenan en ningún otro sitio.

Sin embargo, las bases de datos de expedientes de seguridad son de gran utilidad y cuentan hoy en día con un enorme volumen de *bugs* documentados.

Secunia

Por poner un ejemplo, *Secunia* , que mantiene una de las bases de datos de expedientes de seguridad más respetadas por la comunidad técnica dedicada a seguridad informática, recoge, a día de hoy, *bugs* de más de 20.500 productos de más de 2.300 fabricantes de software.



Bug en software mod_alias de Apache 2.2.x almacenado en la base de expedientes de seguridad en Secunia. Fuente: http://secunia.com/advisories/21490/

Páginas Web

Hoy en día existen múltiples bases de datos, pero quizá las más importantes podrían resumirse en la siguiente lista:

Secunia: http://secunia.com/advisories/

SecurityFocus: http://www.securityfocus.com/bid

Common Vulnerabilities and Exposures: http://cve.mitre.org/cve/

En todas ellas, como se puede ver en la figura anterior, cada *bug* recibe un número de identificación único. En el ejemplo del *bug* de Apache este número es el SA21490, pero este mismo *bug* puede tener otros identificadores en otras bases de datos. En **Security Focus**, este número se conoce como *BugtraqID* y en **Common Vulnerabilites and Exposures** es conocido, simplemente como CVE.

Este código identifica de forma unívoca un *bug* en una base de datos de expedientes de seguridad y a partir de este punto, el fallo de seguridad empieza a ser estudiado para conocer sus implicaciones de seguridad.

1.2. *Exploits*

1.2.1. ¿Qué es un *exploit*?

El *exploit* es un programa creado especialmente para sacar partido de un *bug*. Estos *exploits* pueden ser utilizados para extraer información de una base de datos, para dejar sin conexiones a un servidor web que no gestiona bien la asignación de recursos o incluso para ejecutar un código arbitrario en la máquina donde el software vulnerable está siendo ejecutado.

Poder ejecutar un código arbitrario significaría que un atacante podría conseguir la ejecución del código o el programa que deseara, como por ejemplo una simple calculadora en Windows (muy usada en las pruebas de concepto por ser algo inocuo).

Nos vamos a centrar en las formas en la que los *exploits* pueden ser creados para saber hasta dónde pueden ser utilizados. Como ya se ha dicho, estos *exploits* podrán ser utilizados para elevar privilegios en la máquina, extraer información de la base de datos, hacer que un sistema deje de funcionar o ejecutar un código remoto en una máquina a través de Internet. Todos y cada uno de ellos tendrán importancia en su entorno.

Exploit local

Un ejemplo famoso de *exploit* local es el que permitió ejecutar código en la *consola Wii*. Esta consola viene cerrada por defecto para la ejecución sólo de los juegos autorizados para ella. El *exploit* fue publicado en junio de 2008 y es conocido como el *Twilight Hack*, debido a que el fallo se produce en el juego *The Legend of Zelda: Twilight Princess*. En este juego el protagonista va acompañado de un caballo llamado *Epona*, detalle que no tendría mayor importancia si no fuera porque este nombre aparece dentro de los ficheros de las partidas salvadas. Los miembros del grupo *Team Twiizers* descubrieron que si cambiaban el nombre del caballo por uno especialmente modificado, podían ejecutar sus propios juegos en la *Wii*. Este *exploit* permite, a todo aquel que tenga el juego *The Legend of Zelda: Twilight Princess* ejecutar cualquier programa en la consola *Wii*.

Exploits remotos

Otros ejemplos de *exploits*, en este caso remotos, son los que afectaron a distintos productos de Microsoft a inicios del siglo XXI. *Exploits* como *Code Red* o *SQL Slammer*, ambos gusanos que se aprovechaban de *bugs* explotables remotamente, se hicieron muy famosos por el número de máquinas que afectaron.

Gusano

Un gusano es un virus que, además de infectar el sistema y realizar una serie de acciones, intenta infectar a otros equipos que se encuentren próximos a él mediante la misma vulnerabilidad. Pueden llegar a producir efectos devastadores si el fallo del que se aprovechan es un fallo para el que no existe parche durante largo tiempo.

El primero utilizaba una vulnerabilidad en el servicio *IIS* para lograr ejecutar código en memoria y el segundo se aprovecha de un fallo en la configuración por defecto de la cuenta *sa*, con privilegios administrativos, en los servidores SQL Server.

Podemos encontrar decenas de *exploits* que aprovechan los errores de programación del lector de archivos pdf. Por su gran difusión, se está volviendo el centro de atención de aquellos que desarrollan *exploits* para poder hacerse con el control de las máquinas en que se ejecutan. Podemos encontrar ficheros aparentemente normales en formato PDF que han sigo modificados expresamente para explotar alguna de las vulnerabilidades del lector de Adobe. En abril del 2009 Adobe reconoce un nuevo *0-day* del Acrobat, una vulnerabilidad muy grave de la que se saben todos los detalles. Al ser *0-day*, el *exploit* es público, por lo que se podía explotar y ejecutar código en el sistema. El fallo estaba en la función *getAnnots* y permite a un atacante, de forma sencilla, ejecutar código en el sistema de forma remota. Creando un documento PDF con una anotación y añadiendo un script al PDF a través de la función OpenAction, ya tenia acceso a ejecutar código arbitrario.

Lo mismo ocurre en algunas versiones del tratador de textos MS Word, en el que se puede ejecutar código en las macros.

El software libre tampoco se salva de estos errores; son conocidos los problemas de seguridad que han existido en los programas que se liberan con licencias libres.

Todo el software está expuesto a tener fallos, ya que los desarrolladores pueden cometer fallos de programación por descuido. Es muy fácil dejar de hacer alguna comprobación en una de las miles o millones de líneas de código de un programa.

1.2.2. Tipos de *exploits*

Dentro de los *exploits* hay que diferenciar entre los *exploits* día cero y el resto. Un *exploit* de día cero es aquel para el que no hay un parche creado que solucione el problema.

El ciclo normal de una vulnerabilidad comienza con el descubrimiento del *bug*. Si este *bug* es descubierto por la compañía dueña del software, es probable que no salga a la luz hasta que esté disponible un parche que arregle el software. Esto es muy común en las grandes empresas de software como *Microsoft*, donde se aplica el principio de *responsible disclosure*, es decir, no se publica nada que pueda afectar a la seguridad de sus clientes.

En los entornos en los que no se conoce el *bug* hasta que sale el parche, los investigadores de seguridad realizan procesos de ingeniería inversa sobre el parche para descubrir el *bug* que arregla. Así, viendo lo que arregla saben lo que está mal y generan un *exploit* que saque partido de ese fallo en los equipos que no estén parcheados.

En los entornos de *full disclosure*, el *bug* se hace público en Internet al mismo tiempo que se descubre y la carrera entre el fabricante de software y los creadores del *exploit* puede ser ganada por uno u otro. Algunos llaman a estos *exploits* de día 1, pues se conoce el fallo y no hay parche, pero aún no hay *exploit*.

Los *exploits* de día cero son los más peligrosos para la industria del software. Se trata de fallos de seguridad que sólo conocen una o varias personas y que se pueden explotar en todos los sistemas, pues el fabricante del software desconoce esa vulnerabilidad. Estos son los más cotizados y los que tratan de ser captados por la industria del *malware* y las empresas de seguridad.

1.2.3. Creación de *exploits*

Este es el objetivo del curso, comprender cómo un simple nombre de un caballo puede provocar un fallo de seguridad en un sistema, conocer cuáles son las técnicas para la generación de *exploits* y cuáles son los ejemplos más típicos cuando hablamos de software de plataforma.

Como introducción a la creación de *exploits*, vamos a entender primero cómo trabaja internamente un programa en el ordenador, cómo almacena de forma básica los valores que maneja un programa y también cómo una mala comprobación de los datos de entrada puede generar un comportamiento inseguro del programa.

1.2.4. Memoria del proceso

En la creación de un programa en un lenguaje de alto nivel, por ejemplo en lenguaje C, se hace un uso intensivo de llamadas a funciones. Estas funciones permiten al programador abstraerse de la tarea (por ejemplo, para hacer que un texto sea impreso por pantalla, con la simple invocación de la función *printf* consigue esta acción). Cada una de las llamadas a estas funciones genera la creación de un subproceso, el paso de parámetros entre el proceso principal del programa que llama y el subproceso llamado, la entrega del control del programa principal a la función llamada, la ejecución del código de la función, la entrega de resultados del subproceso llamado al proceso principal y el retorno del control de ejecución al proceso que invocó la función.

Es decir, una simple llamada como `printf("Hola")` generaría la creación de un nuevo proceso con el código de la función *`printf`*, el traspaso de la dirección del proceso principal, donde la función `printf` debe devolver el

control una vez terminada su ejecución, la creación de un espació de memoria para pasar el/los parámetros (en este caso la cadena ASCII `Hola`) y el traspaso del control de ejecución a la dirección donde empieza el código de la función.

La función accederá a los parámetros, ejecutará el código, creará un valor de respuesta de la función (que en este caso es un código de estado de error) y una vez terminada la ejecución devolverá el control a la dirección de memoria que le ha dejado el proceso principal. Todo este intercambio de datos se va a realizar en la memoria, dentro de unas estructuras especiales.

Cuando se ejecuta un proceso dentro de un sistema operativo, este va a tener dos partes distintas en memoria. Una parte estará formada por variables y valores que el sistema configura en la creación del proceso y la otra será la zona de memoria propia, que se utilizará para guardar las variables globales y las variables locales. Será dentro de esta zona donde se creen los subprocesos.

Cada subproceso creado por la invocación de una función tendrá una zona de memoria propia llamada pila (o *snack*). Esta zona de memoria será accesible tanto por el proceso principal como por el subproceso. Esta zona de memoria recibe el nombre de pila por el sistema de gestión que utiliza, basado en filosofía LIFO (*Last In First Out)*, o lo que es lo mismo, el último que entra es el primero que sale.

Como se verá, no comprobar el tamaño de los valores antes de ser asignados a las variables puede permitir que se sobrescriban valores de variables. Esto dará lugar no sólo a la sobrescritura de variables sino también la introducción de código y la sobrescritura de punteros de ejecución, permitiendo llegar a ejecutar código arbitrario en la máquina.

En el ejemplo anterior del *exploit* local para ejecutar código en la consola *Wii* se utilizó algo similar con el nombre del caballo Epona. El *exploit* cambia el nombre del caballo por un valor que sobrescribe la memoria de la consola de tal manera que permite llamar a cualquier programa que se desee.

¿Y es esto posible en todos los programas? No, rotundamente no. Estos errores son comunes en lenguajes de programación como C y C++, lenguajes que no incorporan por defecto ningún tipo de protección frente a la sobrescritura de la memoria. Existen, eso sí, funciones seguras en los lenguajes C y C++ que comprueban el tamaño de las variables y, que de usarse, evitarían los fallos de desbordamiento.

Además de estas protecciones veremos que los lenguajes, los sistemas operativos y los compiladores más modernos implementan mecanismos de protección contra este tipo de vulnerabilidades. Veremos todas estas protecciones en los últimos módulos.

1.2.5. El mercado del *exploit*

La evolución en los sistemas de seguridad informáticos, como es bien sabido, ha llegado a todos los niveles, incluido el del sector comercial. Atrás van quedando los días en los que toda la información era considerada pública, incluida la concerniente a la seguridad. Los investigadores hacían públicos sus trabajos por el mero hecho de su reconocimiento general. En cambio hoy en día, para algunos investigadores de vulnerabilidades el lucro y el dinero son motivaciones tan importantes o incluso más que el reconocimiento de aparecer en uno u otro sitio por haber hecho el descubrimiento de un *bug* en un software.

Alrededor del sector de la seguridad de las vulnerabilidades y los *exploits* se ha generado un mercado de compra-venta bastante significativo, con cifras en ocasiones mareantes y en el que los productos se cotizan de igual forma que un valor en una bolsa cualquiera.

El descubrimiento de un fallo de seguridad y la salida del consiguiente programa que lo aprovecha abren numerosas perspectivas para su utilización: *spammers*, timadores, extorsionadores o espías tecnológicos parecen formar un colectivo sumamente interesado en la adquisición de estas piezas tecnológicas. El uso de redes *botnet* o la toma de control de determinadas máquinas para realizar un ataque y luego eliminar cualquier evidencia suponen una tentación muy interesante que invitan a invertir en la tecnología de las vulnerabilidades.

Botnet

Como botnet se conoce al conjunto de software que se instala en el ordenador mediante algún troyano y que permite controlar el ordenador remotamente. Es utilizado para enviar SPAM o para realizar ataques masivos contra servidores.

No obstante, conscientes de la necesidad de premiar económicamente a los descubridores de los *bugs* importantes, las empresas del sector de la seguridad realizan sonados pagos para hacerse con tan preciado botín y evitar los ataques antes de que aparezcan. Una empresa puede pujar por la compra de uno de los paquetes en venta pensando en mejoras de sus sistemas de auditoría, su posicionamiento en el mercado o simplemente movida por temas de competencia.

Mercado negro de *exploits*

El negocio de la venta de los *exploits*

Un ejemplo bastante interesante del lucrativo negocio de la venta de *exploits* fue el que sacó a la luz *Trend-Micro* en el año 2006. Analistas de esta compañía, que se infiltraron en el mundo *underground* de las redes *hacker*, detectaron la venta de un *exploit* para Windows Vista por la nada despreciable cifra de 50.000 dólares. Un año antes, la compañía *Kaspersky* hacía pública también la posible venta de una vulnerabilidad de día cero para el formato de fichero WMF por parte de un colectivo *hacker* ruso al precio de 4.000 dólares. Dicha vulnerabilidad permitía la ejecución remota de código a través del navegador Intenet Explorer.

Aunque las cifras aquí barajadas pueden parecer significativamente altas, no son ni más ni menos que una apreciación real del mercado negro que se mueve en dicho sector y el dinero que genera actualmente el mercado del *malware*. Evidentemente, la procedencia del dinero no será todo lo legal que uno desearía y como tal, todo este negocio se encuentra fuera del amparo legal establecido.

Mercado de empresas de seguridad

Por el contrario, existe otro tipo de mercado más lícito para evitar estas prácticas irregulares, como las iniciativas promovidas por las empresas de seguridad informática. Incentivan la publicación de fallos, ofreciendo una recompensa económica al fruto de toda una investigación. En este sentido podemos encontrar la iniciativa *Zero Day Initiative (ZDI)* fundada por *TippingPoint*. Consiste en un programa para recompensar a los investigadores de seguridad por divulgar vulnerabilidades responsablemente. Los beneficios económicos obtenidos a través de esta iniciativa no serán tan altos como los que se pueden conseguir en el mercado negro, pero el dinero así ganado es totalmente legal y como tal está libre de cualquier complicación.

Beneficios del programa *Zero Day Initiative*

Zero Day Initiative

A través de la url http://www.zerodayinitiative.com/about/benefits/ se pueden conocer cuáles son los beneficios a obtener por dar a conocer una vulnerabilidad descubierta. Los *bugs* descubiertos son valorados y cuantificados mediante una serie de parámetros que miden la criticidad, el impacto o la reproductibilidad del ataque. Proporcionan una serie de puntos que pueden ser canjeados por dinero metálico o usados para inscribirse y viajar a conferencias de reconocido prestigio en el sector de la seguridad, por ejemplo la *DEFCON* y la *BlackHat*.

iDefense Labs

A través de su programa VCP (*Vulnerability Contributor Program*), que comenzó su andadura en el año 2002, también incentivan económicamente la publicación de fallos de seguridad y las pruebas de concepto de los mismos con un valor máximo de 15.000 dólares. Además, anualmente presentan un certamen donde las vulnerabilidades del programa más significativas del año son premiadas con cuantías que van desde los 5.000 hasta los 50.000 dólares.

Pack Agora

La empresa Gleg Ltd. adquirió en el año 2007 un paquete de *exploits* de día 0 generados por Argeniss, dentro de los cuales destacaba la explotación de bases de datos *Oracle* a través de uno de los elementos del paquete adquirido. Este paquete, junto con otros módulos de Gleg Ltd., fue comercializado posteriormente en el año 2008 en el *Pack Agora*, que podía ser comprado para realizar auditorías y tests de seguridad.

WabiSabiLabi

Otro ejemplo interesante de este mercado emergente lo propuso la firma suiza WabiSabiLabi, que en el año 2007, en base a un estudio de mercado que realizó, decidió plantear un sistema similar a eBay pero dentro del mercado de las vulnerabilidades. Su periplo comenzó con vulnerabilidades para Yahoo Messenger y Squirrelmail pero en apariencia el estudio de mercado no fue totalmente satisfactorio o bien la idea no fue muy bien planteada, puesto que actualmente el servicio ya no se encuentra disponible.

1.2.6. Valoración del *exploit*

Dentro del mercado de compra-venta, independientemente de lo legal o ilegal que pueda ser, el precio que puede alcanzar una vulnerabilidad o un *exploit* viene prefijado por una serie de parámetros, entre los que la popularidad de la aplicación o el sistema afectado es el más importante. Como todo el mundo reconocerá, un fallo de seguridad en Acrobat Reader tiene más valor en el mercado que el detectado en una aplicación de poco uso o de difícil repercusión. Asímismo, no tendrá la misma graduación una vulnerabilidad de un sistema operativo moderno, por ejemplo Windows Vista, como la que pudiera ofrecer en la actualidad un fallo de seguridad en Windows 2000.

Otro de los factores importantes de un *exploit* es el nivel de aplicación que se puede llegar a ofrecer o la facilidad de uso del mismo. No es lo mismo un *exploit* que permita la ejecución remota de código con elevación de privilegios que otro en el que la explotación haya que realizarla localmente y que para ello, además, ya deban tenerse determinados privilegios adquiridos.

Hoy en día todo tiene un precio, aunque su cuantificación se revaloriza o cae en función del momento. No obstante, determinados elementos pueden considerarse de forma constante como los *top* dentro de los más pagados y, cómo no, de los más demandados. Fallos de seguridad en servicios archiconocidos como *Hotmail*, *Gmail* o *Messenger*, los últimos sistemas operativos de Microsoft o aplicaciones tan comunes como *Acrobat Reader*, *Macromedia Flash* y los servidores *Apache*, constituyen la punta de lanza de un "sector comercial" en constante auge.

1.2.7. Estuctura del *exploit*

En este libro vamos a mostrar dos partes bien diferenciadas. La primera estará centrada en las herramientas y los conceptos necesarios para poder entender la segunda, que se centrará más en los diferentes *exploits* y vulnerabilidades de los sistemas.

Comenzaremos, entonces, con una visión no muy detallada del lenguaje máquina y del ensamblador, que nos permitirá comprender cómo se puede variar según nuestra voluntad el comportamiento de un software ya existente.

En esta parte cobran mucha importancia los registros internos del ordenador, ya que sobre ellos van a ir las direcciones de memoria que podremos variar a posteriori. Veremos, pues, los diferentes registros y qué se guarda en ellos usualmente.

A continuación veremos la descripción de los operandos de ensamblador, que nos permitirán cambiar las direcciones de memoria de los registros de retorno de las funciones. Seguiremos con los conceptos de la ejecución de procesos y eso nos permitirá ver dónde y cómo se almacenan las variables, lo que aprovecharemos para dar algún ejemplo de variación de los valores.

Para acabar con la primera parte, veremos las herramientas más comunes que se usan para seguir el comportamiento de un programa y poder cambiarlo. Estos *debuggers* nos permitirán entender qué hace el programa en cada momento y cómo lo podemos modificar para que se comporte como nosotros queramos.

Una vez tengamos claros los conceptos y las herramientas a utilizar, se describirán los *exploits* de programación existentes.

Veremos cómo se hace para variar el comportamiento de un programa y cómo podemos generar un *exploit* de forma automática, usando herramientas ya existentes que generan los programas que explotan una vulnerabilidad de un determinado sistema operativo.

Además de ver cómo se puede atacar a los sistemas, veremos cómo se protegen las nuevas versiones de los sistemas operativos de estos ataques clásicos. La concienciación generalizada por parte de los desarrolladores de software de que la mala programación genera enormes pérdidas de dinero hace que cada vez existan menos vulnerabilidades y los sistemas operativos sean cada vez más seguros. Esto lo demuestra la poca cantidad de vulnerabilidades dadas a conocer en relación con las últimas versiones de los sistemas actuales comparadas con las anteriores.

2. Gestión de memoria

Contenido complementario

Intel. Disponible en: http://www.intel.com/

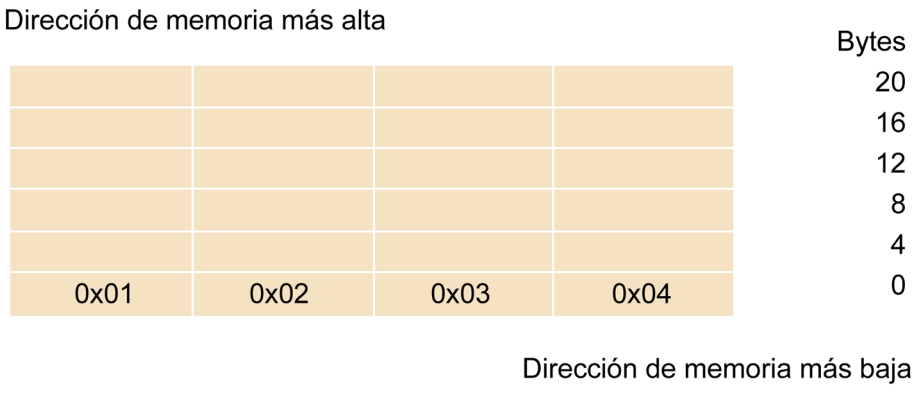
Para crear o comprender un *exploit* es muy importante conocer profundamente la arquitectura y el funcionamiento de la máquina que se quiere atacar. Muchos *exploits* aprovechan el funcionamiento normal de un sistema para obtener control sobre él, aprovechando errores que el programador de la aplicación/sistema, ha cometido. Como caso de estudio nos centraremos en la arquitectura de los procesadores de Intel. Otros procesadores o arquitecturas hardware pueden procesar la información de manera diferente. Así pues, un *exploit* es totalmente dependiente de la arquitectura del sistema a atacar.

2.1. Orden de escritura de bits y bytes en memoria

Existen varias convenciones para escribir datos en memoria. Las más conocidas son las escrituras en *little-endian* y *big-endian*.

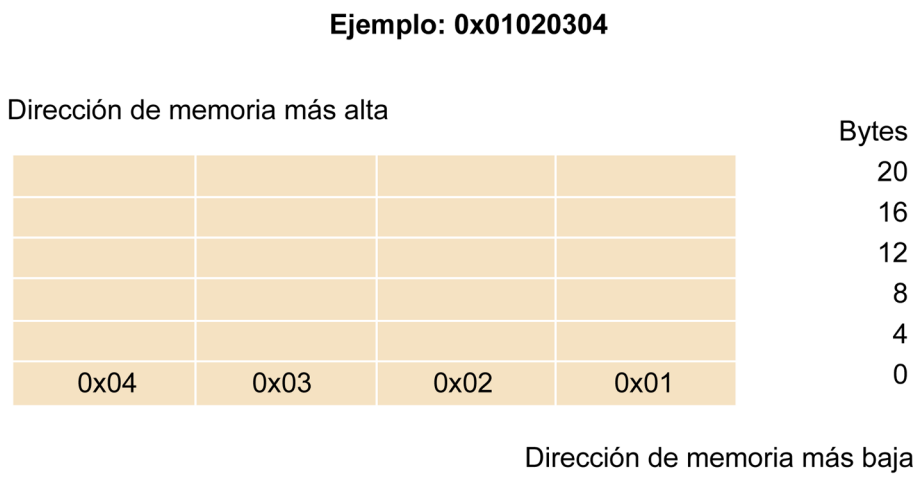
Little-endian escribe el byte menos significativo (LSB – *least significant byte*) en la dirección de memoria más baja. En el siguiente ejemplo, se escribe en memoria un dato siguiendo este sistema.

Ejemplo: 0x01020304



Little-endian

Big-endian, al contrario de *little-endian*, escribe el byte más significativo (MSB – *most significant byte*) en la dirección de memoria más baja.



Big-endian

Los sistemas basados en la arquitectura Intel (x86 hasta IA-32 e Intel 64) funcionan usando el modelo *little-endian*.

2.2. Segmentos

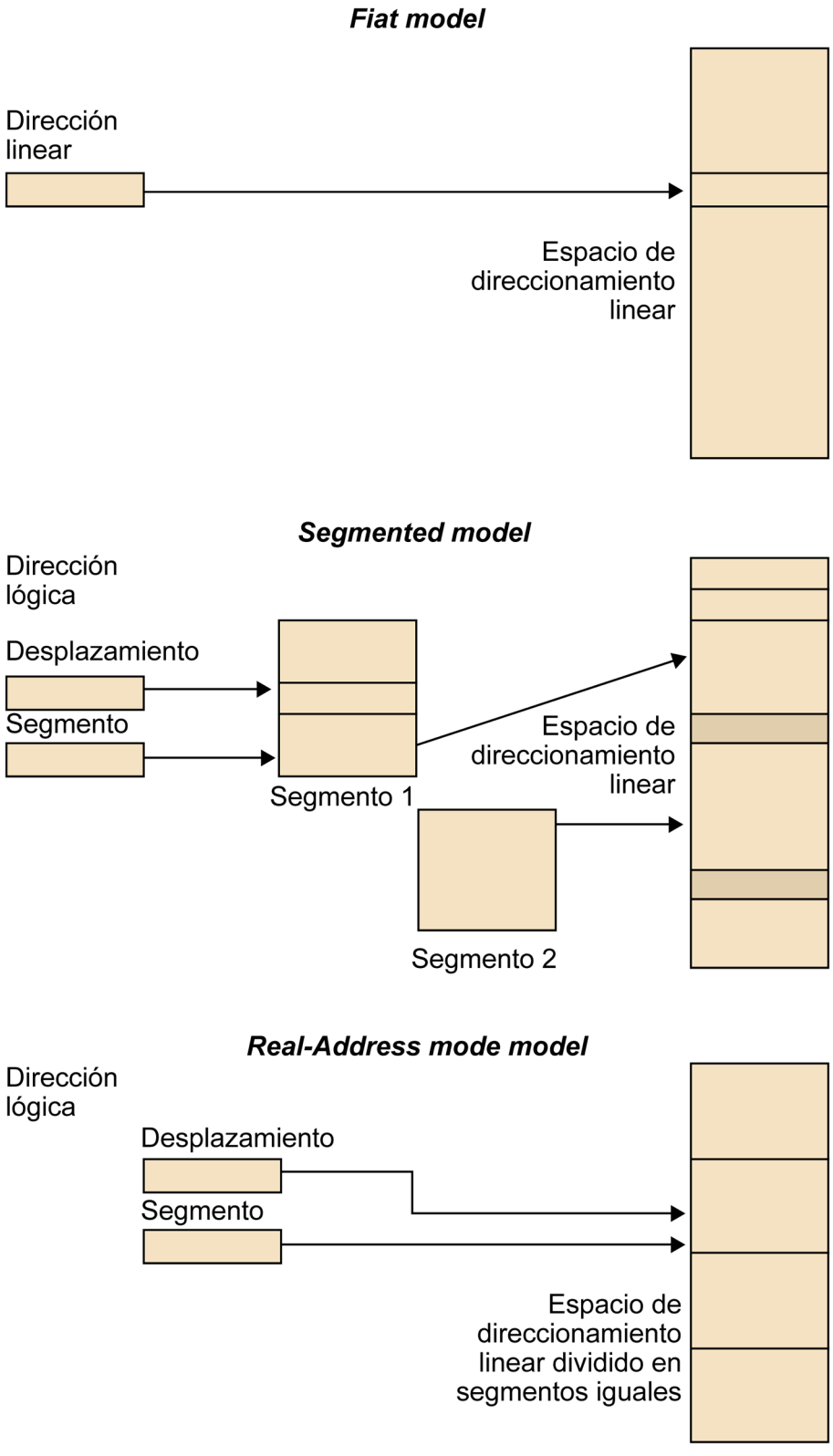
Cuando un programa necesita acceder a memoria, no accede directamente a la memoria física sino que usa uno de los modelos de gestión de memoria que el procesador pone a su alcance. En los sistemas basados en x86 encontramos 3 modelos de memoria posibles:

- *Flat memory model*. Este modelo muestra la memoria al programa como un espacio de direccionamiento único y contiguo. Todos los datos necesarios para la ejecución del programa se encuentran en el mismo espacio de direccionamiento (datos, pila, programa).

- *Segmented memory model*. Este modelo es el más usado por los programas y muestra la memoria como un grupo de espacios de direccionamiento independientes llamados segmentos. Así pues, existen segmentos distintos para usos distintos: datos, pila, código. Para acceder a un byte de un segmento los programas hacen una petición de una dirección lógica. Esta dirección contiene el segmento que se quiere acceder más un desplazamiento dentro de él (*offset*). Internamente el procesador hace la conversión de la dirección lógica a la dirección final dentro del espacio de direccionamiento del procesador.
  El uso de segmentos aumenta la fiabilidad de los sistemas, evitando así que la pila crezca hasta sobrescribir el código del programa, entre otras cosas.

- *Real-address mode memory model*. Este modelo existe para ofrecer compatibilidad con programas escritos para procesadores 8086, permitiendo así su uso en procesadores más modernos. Con los modelos de memoria *flat* o *segmented* el espacio de direccionamiento linear se usa a través del espacio de direccionamiento físico del procesador, directamente o mediante pagi-

nación. Si la paginación no se usa, cada dirección solicitada por el procesador tiene correspondencia directa con una dirección física.



Modelos de gestión de memoria

Si se usa paginación, el espacio de direccionamiento linear se divide en fragmentos llamados "páginas" y éstas pasan a formar parte de la "memoria virtual". Estas páginas de datos son gestionadas de manera transparente por el

sistema. Las aplicaciones sólo ven un espacio de direccionamiento linear al que tienen acceso, pero este se gestiona de un modo distinto al que las aplicaciones ven.

2.3. Registros básicos para la ejecución de programas

Uno de los elementos más importantes para gestionar el sistema son los registros.

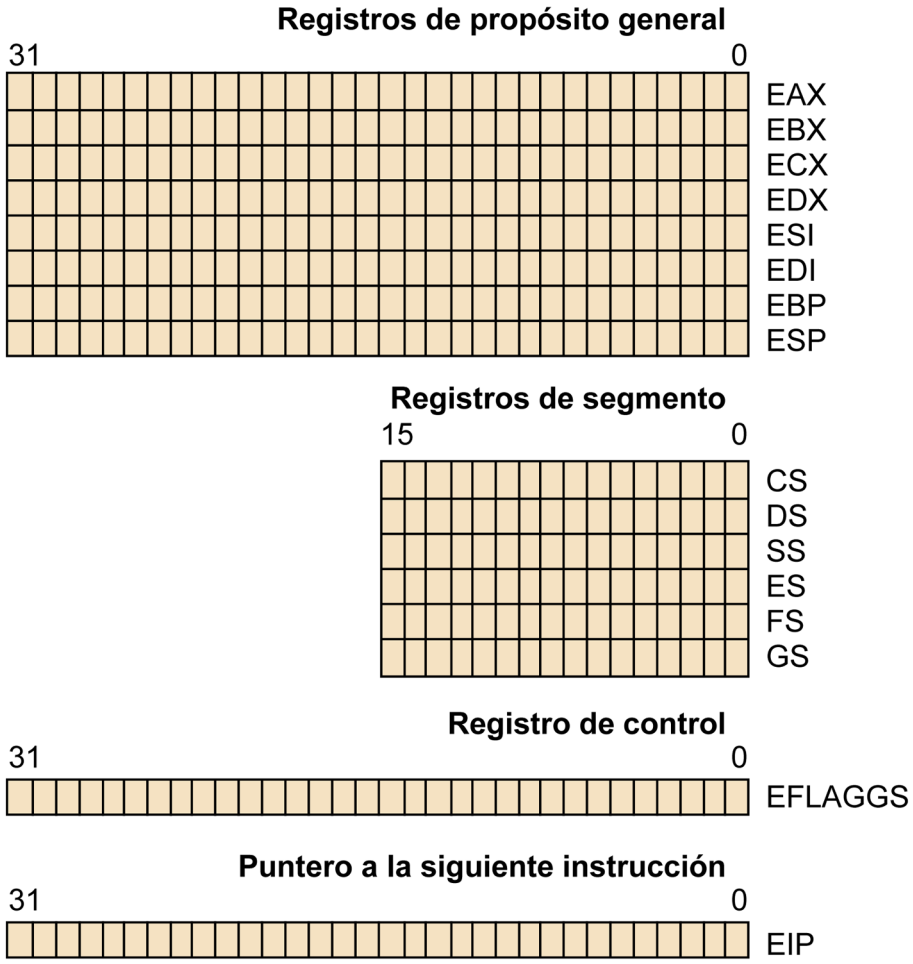
Estos registros básicos en una arquitectura x86 se pueden agrupar en las siguientes categorías:

- Registros de propósito general. Son ocho registros de propósito general para guardar datos y punteros.

- Registros de segmento. Existen hasta un total de 6 selectores de segmentos.

- Registro *EFLAGS*. Es un registro especial encargado de llevar el control de ciertas operaciones en la ejecución de un programa, y puede tener cierta incidencia sobre el procesador.

- Registro *EIP*. Este registro contiene siempre un puntero a la siguiente instrucción a ejecutar por el procesador. Toma especial interés cuando se está atacando una máquina mediante un *exploit*. El objetivo principal es cambiar este registro y asignarle un puntero a una dirección de memoria en donde hayamos podido cargar nuestro código.



Registros del sistema

2.3.1. Registros de propósito general

Los registros de propósito general son los siguientes: EAX, EBX, ECX, EDX, ESI, EDI, EBP y ESP. Se usan para hacer cálculos numéricos, cálculos de direcciones de memoria y también como punteros a memoria.

Aunque todos se pueden usar de un modo general, la mayoría de ellos tiene usos predefinidos. Por ejemplo, el registro ESP se usa para controlar la pila (*stack*). Así pues, es altamente recomendable (u obligatorio) no usarlo para ninguna otra función. Otros registros con utilidades preestablecidas son el EBP (para el control de los contextos de ejecución y acceso a los parámetros en la pila), o los ECX, ESI y EDI que se utilizan en las operaciones con cadenas de caracteres (*strings*).

Los usos especiales de cada registro son los siguientes:

- EAX. También llamado acumulador. Sirve como recipiente para guardar el resultado de muchas operaciones. También se usa para retornar valores en llamadas a funciones.

- EBX. Puntero a datos usando el segmento de datos (DS).

- ECX. También llamado contador. Se suele usar como contador en bucles.

- EDX. Puntero de entrada/salida.

- ESI. Puntero a datos usando en segmento de datos (DS). Puntero a los datos fuente para operaciones relacionadas con *strings*.

- EDI. Puntero a datos usando el segmento de datos (DS). Puntero a la dirección destino para operaciones relacionadas con *strings*.

- ESP. Puntero de pila (*Stack Pointer*). Controla el nivel de la pila.

- EBP. Puntero a la pila (*Base Pointer*). Apunta a datos en la pila.

Todos estos registros son de 32 bits, tal como se muestra en el gráfico anterior. Aún así, para mantener la compatibilidad con sistemas antiguos, los mismos registros existen en una arquitectura de 16 bits, pero sus nombres cambian, pierden la *E* que los precede. En la versión de 16 bits los nombres de esos registros serían: AX, BX, CX, DX, SI, DI, SP BP. Aunque el nombre cambia ligeramente, las funciones asignadas siguen siendo las mismas. Además de esta nomenclatura, se puede acceder por separado al byte alto y bajo de los registros AX, BX, CX y DX cambiando la X por la H de *high* ('parte alta') o la L de *low* ('parte baja').

- Registros de 32 bits (*doubleword registers*): EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP.
- Registros de 16 bits (*word registers*): AX, BX, CX, DX, SI, DI, SP, BP.
- Acceso a bytes de los registros AX, BX, CX y DX (*byte registers*): AH, AL, BH, BL, CH, CL, DH y DL.

2.3.2. Registros de segmentos

Los registros de segmentos son seis. El tamaño de todos ellos es de 16 bits y se usan para crear segmentos en memoria que serán destinados a usos distintos. Estos registros son: CS (*Code Segment*), DS (*Data Segment*), SS (*Stack Segment*), ES (*Extra Segment*), FS y GS.

Los cuatro segmentos de memoria principales CS, DS, SS y ES tienen un uso ya establecido.

- CS. Segmento en el que se almacena el programa en ejecución. El procesador obtiene las instrucciones a ejecutar utilizando este segmento y el registro EIP. El registro EIP contiene el desplazamiento desde el principio de segmento definido por CS, hasta la siguiente instrucción a ejecutar. El registro CS no puede ser cambiado de manera explícita por un programa. Este registro es gestionado de manera implícita por el sistema.

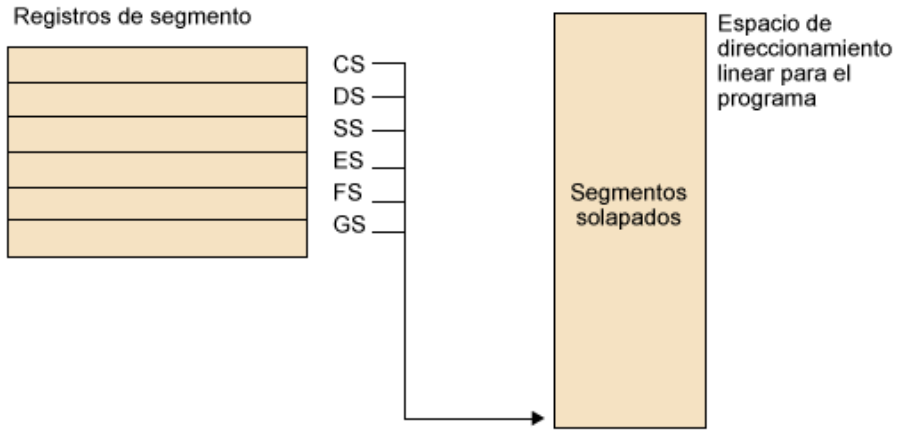
- DS. Se guardan los datos que el programa utiliza.

- SS. En este segmento se crea la pila de datos, necesaria para la ejecución de todos los programas. Todas las operaciones que necesitan la pila usan este segmento para acceder a ésta. Este registro sí puede ser cambiado por un programa. Esto permite la creación de varias pilas para un programa y poder cambiar entre ellas según se requiera.

- ES. Segmento de datos extra para poder almacenar más datos en memoria.

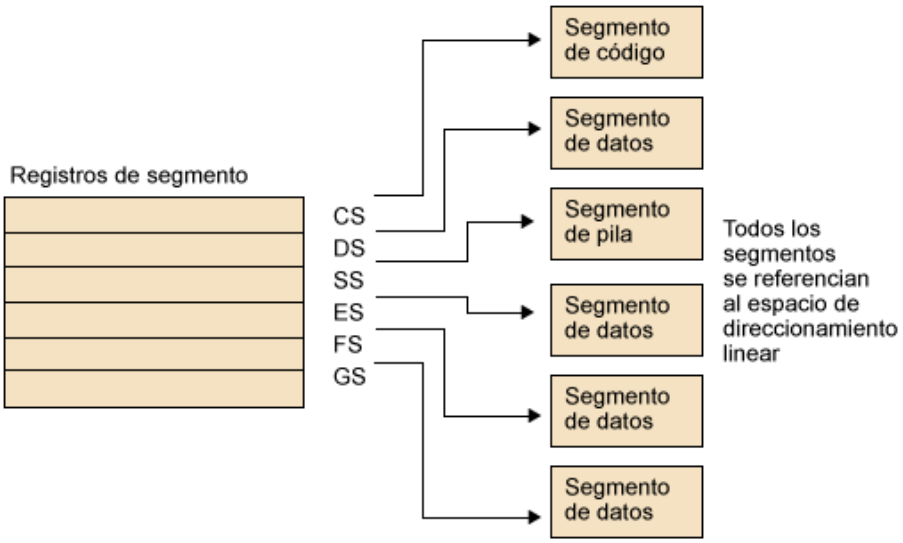
Los segmentos FS y GS se introdujeron más tarde en la arquitectura x86 y se utilizan como segmentos de datos adicionales al igual que el segmento ES.

La utilización de los segmentos depende del modelo de memoria utilizado. Si se utiliza el modelo de memoria *flat*, entonces todos los segmentos apuntarán al mismo segmento de memoria, resultando un sistema con todos los segmentos superpuestos.



Registros de segmento usando el modelo "flat"

Si se utiliza un modelo de memoria segmentado, entonces se pueden crear segmentos distintos que apunten a segmentos de memoria totalmente separados. De esta manera un programa puede acceder a seis segmentos distintos en el espacio de memoria. En el caso de querer acceder a un segmento del que no dispone, primero tiene que cargar el valor apropiado en el registro de segmento y después acceder a los datos del nuevo segmento.



Registros de segmento utilizando el modelo "segmented"

2.3.3. Registro EFLAGS

El registro EFLAGS lleva el control de una serie de indicadores del sistema entre los que hay indicadores de estado, de control y de sistema. Cada bit de este registro indica una situación concreta. Durante la ejecución de un programa este registro va modificando sus bits dependiendo de los resultados de las distintas operaciones que se ejecutan en el programa. Estos bits pueden ser leídos e interpretados para su uso durante la ejecución del programa. Hay varias instrucciones que permiten mover el contenido de este registro a la pila o al registro EAX. Una vez quee se tiene el valor en uno de esos dos lugares se puede inspeccionar su contenido. No se puede cambiar el valor del registro EFLAGS de manera directa.

Los diferentes bits que se encuentran en este registro se suelen llamar "flags" y los que no están reservados (no se pueden usar) son los siguientes:

- *ID Flag* (ID). *Flag* de sistema.
- *Virtual Interrupt Pending* (VIP). *Flag* de sistema.
- *Virtual Interrupt Flag* (VIF). *Flag* de sistema.
- *Alignment Check* (AC). *Flag* de sistema.
- *Virtual-8086 Code* (VM). *Flag* de sistema.
- *Resume Flag* (RF). *Flag* de sistema.
- *Nested Task* (NT). *Flag* de sistema.
- *I/O Privilege Level* (IOPL). *Flag* de sistema.
- *Overflow Flag* (OF). *Flag* de estado.
- *Direction Flag* (DF). *Flag* de control.
- *Interrupt Enable Flag* (IF). *Flag* de sistema.
- *Trap Flag* (TF). *Flag* de sistema.
- *Sign Flag* (SF). *Flag* de estado.
- *Zero Flag* (ZF). *Flag* de estado.
- *Auxiliary Carry Flag* (AF). *Flag* de estado.

- *Parity Flag* (PF). *Flag* de estado.
- *Carry Flag* (CF). *Flag* de estado.

Flags de estado

Estos *flags* dan indicaciones sobre el resultado de operaciones aritméticas como pueden ser las instrucciones ADD o SUB. Los usos de los diferentes bits de estado son los siguientes:

- *Carry Flag* (CF). Indica si la operación ha generado un resultado mayor que el registro resultado es capaz de contener.

- *Parity Flag* (PF). Indica si el bit de menor peso del resultado es un uno. Detección de resultados impares.

- *Adjust Flag* (AF). Indica si la operación ha generado un resultado mayor que el bit 3 del resultado.

- *Zero Flag* (FG). Indica si el resultado es cero.

- *Sign Flag* (SF). Adquiere el valor del bit de mayor peso del resultado de la operación. En operaciones con enteros 0 indica un número positivo y 1 indica un número negativo.

- *Overflow Falg* (OF). En operaciones con enteros este bit indica si el resultado es demasiado grande (números positivos) o demasiado pequeño (números negativos) para caber en el contenedor de destino.

Flags de control

El único *flag* en esta categoría es el *Direction Flag* (DF). Este *flag* indica en qué dirección se van a procesar las instrucciones relacionadas con *strings*. Esto significa si las operaciones con *strings* van a ser tratadas con auto-incrementos (bit a 0 – de direcciones de memoria bajas a direcciones altas), o con auto-decrementos (bit a 1 – de direcciones de memoria altas a direcciones bajas). Hay dos instrucciones que cambian este *flag*: STD (bit a 1), CLD (bit a 0).

Flags de sistema

Estos *flags* controlan el comportamiento del sistema, así que no deben ser cambiados por las aplicaciones. Los distintos *flags* de sistema son los siguientes:

- *Trap Flag* (TF). A uno, activa el trazado (*debug*) paso a paso.

- *Interrupt enable Flag* (IF). A uno, activa la respuesta a las interrupciones. A cero, desactiva las interrupciones.

- *I/O privilege level field* (IOPL). Este campo está compuesto por dos bits e indica el privilegio actual del programa en curso para acceder al espacio de entrada/salida.

- *Nested taks Flag* (NF). Controla el encadenamiento de tareas en interrupciones o llamadas.

- *Resume Flag* (RF). Controla la respuesta del procesador a las excepciones de trazado (*debug*).

- *Virtual-8086 mode flag* (VM). A uno para activar el modo 8086.

- *Alignment check flag* (AF). Controla si se tiene que hacer testeo de alineación de referencias a memoria.

- *Virtual interrupt flag* (VIF). Usado en el control de las interrupciones.

- *Virtual interrupt pending flag* (VIP). A uno indica que hay una interrupción pendiente.

- *Identification falg* (ID). Da soporte para el comando CPUID.

Registro EIP

Este registro controla la ejecución del programa. Contiene un desplazamiento en memoria, contado teniendo en cuenta el segmento de código (CS) que indica cuál va a ser la próxima instrucción a ejecutar. Este registro no puede ser alterado directamente, pero hay varias operaciones que tienen impacto sobre él: JMP, Jcc, CALL, RET y IRET (se verán más adelante).

Una manera de acceder al valor del registro EIP es ejecutar una instrucción CALL (la cual escribe en la pila el valor de EIP entre otras cosas), y luego leer el valor de EIP en la propia pila. Aún así, será el valor de retorno de la instrucción CALL, y nunca el valor actual del registro EIP. Se estudia este proceso en detalle más adelante, ya que los *exploits* hacen servir estos registros para saltar de función.

2.4. Utilización de la pila

La pila es uno de los recursos importantes en el funcionamiento de los sistemas. Es un recurso que debe ser administrado con sumo cuidado ya que de él depende, en parte, el buen funcionamiento del sistema.

La pila es una forma de guardar datos que se basa en una estructura LIFO (*last in, first out*). Así pues los últimos elementos en llegar a la pila, son los primeros que serán sacados. Esta manera de proceder encaja perfectamente con el comportamiento de las estructuras de ejecución de los programas.

Es por ello que la pila se usa para llevar el control de las llamadas a funciones y procedimientos de manera anidada en los programas.

Como ya se ha comentado, en la arquitectura x86 se usan los registros SS *(stack segment*) y SP *(stack pointer*) como registros básicos para acceder a la pila. A estos se debe añadir el registro BP (*base pointer*), que se usa para acceder a datos cuya ubicación está en la pila. BP usa el segmento SS por defecto, al igual que el registro SP. Este último registro se encarga de controlar la parte superior de la pila (*top*).

Informaciones que se guardan en la pila:

- Dirección de retorno. Uno de los datos más importantes que se guardan en la pila es la dirección de retorno de la ejecución del programa. Cuando se llama a una función/procedimiento, la dirección de retorno de la ejecución del programa se guarda en la pila. Después de ejecutar la función o procedimiento, se reanuda la ejecución en el punto donde se había interrumpido.

- Espacio para datos locales. Las variables declaradas de manera local a una función son también guardadas en la pila. Así pues, desaparecen totalmente después de la ejecución.

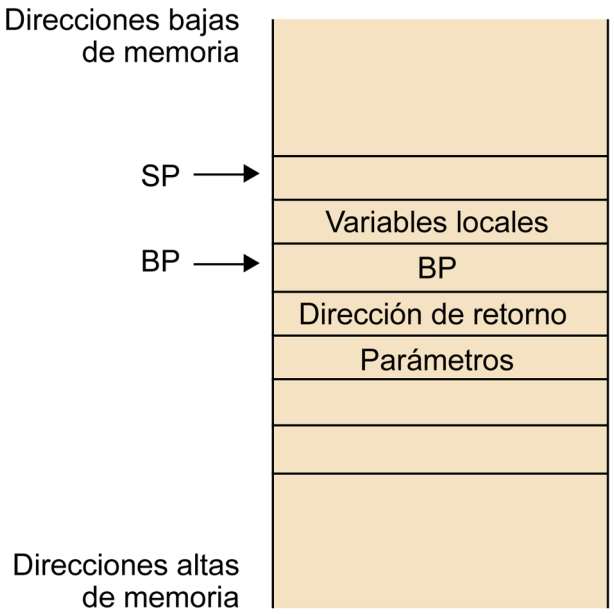
- Paso de parámetros. Los parámetros necesarios para la ejecución de las funciones llamadas también se guardan en la pila.

- Retorno de resultados. La pila también se usa para poder devolver los resultados de las funciones llamadas.

- Estados de la ejecución. En determinadas situaciones también se guardan en la pila datos correspondientes al estado de la ejecución, por ejemplo el registro EFLAGS.

- Espacio para cálculos. A veces se utiliza este recurso para realizar cálculos con distintos operandos.

El esquema de la organización de la pila durante la ejecución de un programa es el siguiente:

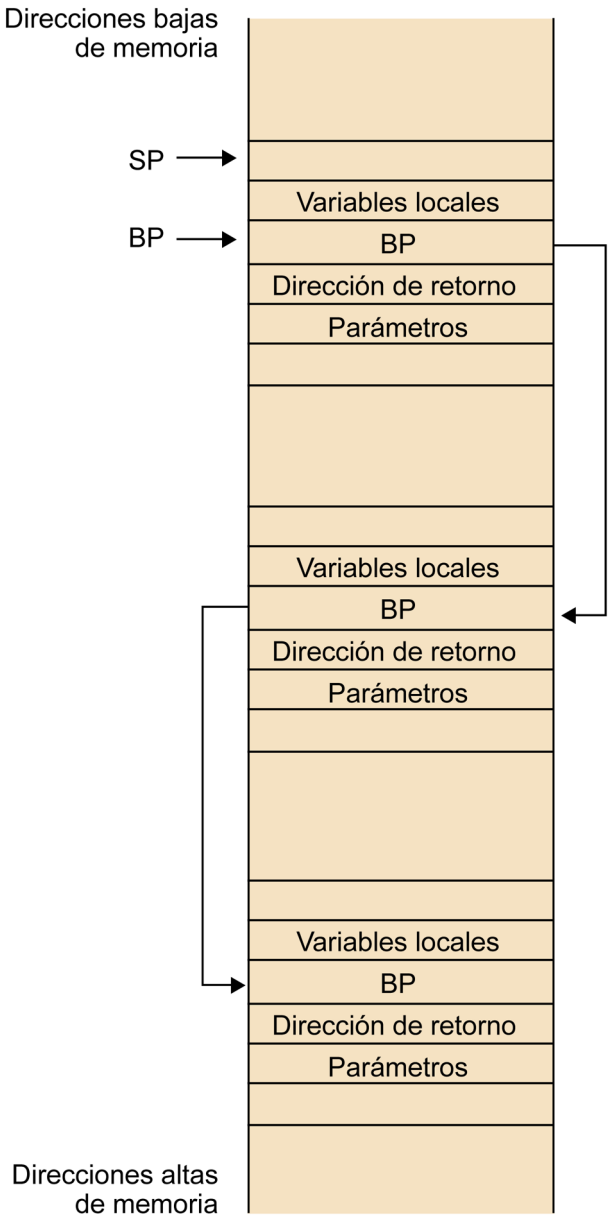


Estructura del contenido de la pila

En el gráfico anterior vemos la estructura del contenido de la pila, más los dos registros que se usan para acceder a los datos que en ella se encuentran, juntamente y de manera transparente con en registro de segmento SS. El registro SP siempre apunta a la parte superior de la pila (*top*), y el registro BP apunta al último valor guardado, que a su vez apunta a su anterior valor guardado en la pila. Esto produce una serie de encadenamientos que sirve para mantener el seguimiento de los diferentes contextos de ejecución de las distintas funciones llamadas.

También se aprecia que la pila empieza en las direcciones altas de memoria y va creciendo hacia las direcciones bajas. Cada vez que se introduce un elemento en la pila, el registro SP se decrementa, aumentando así el tamaño de la pila. Para liberar elementos de la pila el registro SP se incrementa, reduciendo así el tamaño de la pila.

Si se observa como queda la pila después de hacer llamadas sucesivas a funciones o procedimientos, se obtiene una estructura como la siguiente:



Encadenamiento de las estructuras de datos de la pila

Así como el registro SP es un valor volátil (siempre apunta a la parte superior de la pila, que puede ser muy variable), el contenido del registro BP es bastante estable. Es siempre un apuntador a su antiguo valor en la pila, por lo que su utilización es ideal para acceder a valores que se guardan en ella. Este registro se usa sobre todo para acceder a los parámetros que se pasan a las funciones y a las variables locales a las que se les ha reservado espacio en la pila.

3. Conceptos básicos de lenguaje máquina

Todo programa realizado en un lenguaje de alto nivel es transformado para que pueda ser ejecutado por la máquina para la cual se ha hecho. En este proceso se realizan dos pasos importantes: compilación y enlazado *(link)*. Un caso aparte sería la utilización de lenguajes interpretados.

El resultado de realizar estos dos procesos es un programa ejecutable que está escrito en un lenguaje que la máquina es capaz de ejecutar (código máquina). Durante la fase de compilación se realiza la traducción del lenguaje fuente a código máquina y durante la fase de enlazado (*link*) se enlaza ese código final con todas las librerías necesarias para que el programa pueda ser ejecutado. Este enlazado puede ser estático (todas las librerías necesarias pasan a incluirse en el resultado final) o dinámico (el código final queda preparado para enlazarse a las librerías en el momento de la ejecución).

Puesto que es el código máquina el que acaba ejecutándose, es importante conocer unos cuantos detalles acerca de cómo funciona, más concretamente el lenguaje ensamblador.

El código máquina es el lenguaje que el ordenador es capaz de ejecutar directamente, pero es difícil de leer para las personas. Son tan solo códigos en binario (para una mayor comprensión se suelen leer en hexadecimal) que indican al procesador qué es lo que tiene que ir haciendo. Debido a esta dificultad se crearon una serie de etiquetas que describen esos códigos para hacerlos así más inteligibles. Esta serie de etiquetas es lo que se denomina lenguaje ensamblador y es el lenguaje de programación más cercano al lenguaje máquina. El lenguaje ensamblador también necesita ser compilado, pero la traducción a lenguaje máquina suele ser directa. Así pues, es posible seguir la ejecución de un programa en memoria usando herramientas de trazado *(debuggers)* a nivel de lenguaje ensamblador.

Cada procesador tiene un conjunto de instrucciones que es capaz de ejecutar. Estas instrucciones se ejecutan a nivel de registro o memoria. Así pues, existen varios tipos de operaciones que se pueden hacer:

- Instrucciones de transferencia de datos: MOV, PUSH, POP...
- Operaciones binarias entre enteros: ADD, SUB, MUL, DIV...
- Aritmética decimal: DAA, DAS...
- Operaciones lógicas: AND, XOR, OR, NOT...
- Desplazamientos y rotaciones: SHR, SHL, ROR, ROL...
- Operaciones de bits y bytes: BTS, BTR, BSR, TEST...
- Operaciones para el control del programa: JMP, JE, JZ, CALL, RET...
- Operaciones para cadenas de caracteres: MOVS, CMPS, REP...

- Operaciones de entrada/salida: IN, OUT, INS...
- Control de *flags:* STC, CLC, PUSHF, POPF...
- Operaciones de registros de segmentos: LDS, LES, LFS...
- Varias no incluidas en las categorías anteriores: LEA, NOP, CPUID...

Dependiendo del tipo de procesador, habrá otros subconjuntos de operaciones disponibles. Es tarea del compilador usar un determinado subconjunto de instrucciones para tener así un rendimiento óptimo del programa final, si tiene datos acerca de la arquitectura final sobre la que se ejecutará el programa. Posibles subconjuntos de operaciones disponibles:

- x87 FPU. Instrucciones para realizar cálculos con números en coma flotante.
- MMX. Extensiones multimedia.
- SSE. Extensiones para *streaming* SIMD.
- SSE-2. Ampliación de las extensiones SSE para poder realizar operaciones en coma flotante con operandos de 64 bits.
- SSE-3. Mejora del rendimiento de las extensiones SSE.
- SSSE-3. Mejora de rendimiento con enteros.
- SSE-4, SSE-4.1, SSE-4.2. Subconjuntos de operaciones pensados para mejorar el rendimiento en determinados tipos de aplicaciones gráficas.

3.1. Tipos de nomenclaturas

Existen dos tipos de nomenclaturas ampliamente utilizadas cuando se trabaja en ensamblador. Estas dos son:

- Intel
- AT&T[1]

La sintaxis de Intel es la más utilizada en entornos Windows[2] y la de AT&T es muy utilizada en entornos Unix[3]/Linux[4]. Cuando se está trazando un programa, primero hay que ver qué tipo de nomenclatura está siendo usada por la herramienta.

Las dos nomenclaturas tienen diferencias:

- En la sintaxis de Intel primero se escribe el operando destino y después el fuente. Esto es totalmente al revés en la nomenclatura AT&T.
- En la sintaxis AT&T los códigos de operación/instrucciones *(opcodes)* van seguidos de una letra indicando así el tamaño de los operandos (*l* para doble *word*, *w* para *word* y *b* para byte).

(1)**AT&T**. Disponible en: http://www.att.com/

(2)**Microsoft Windows**. Disponible en: http://www.microsoft.com/windows/

(3)**Unix**. Disponible en: http://www.unix.org/

(4)**Linux**. Disponible en: http://www.kernel.org/

- En la sintaxis AT&T los valores van precedidos de un $ y los registros de un %.

- En la sintaxis AT&T las referencias a posiciones a memoria se escriben entre paréntesis, en cambio en la sintaxis de Intel se escriben entre corchetes.

3.2. Operaciones y operandos en ensamblador

Las instrucciones en ensamblador suelen ser sentencias cortas con dos, uno o ningún operando explícito. Muchos de los comandos implican la utilización de registros de manera implícita.

Las operaciones y operandos en lenguaje máquina se suelen leer en hexadecimal, y como ya se ha comentado, tienen una traducción directa en lenguaje ensamblador. La siguiente sentencia está escrita en lenguaje ensamblador y ejecutaría una asignación de un valor (`0x110`) sobre el registro edx:

```
mov $0x110, %edx
```

Como se puede observar, la instrucción está escrita en sintaxis AT&T. Esto indica que el operando destino es el segundo y que el fuente es el primero. Esta sentencia en lenguaje máquina (hexadecimal) sería:

```
ba 10 01 00 00
```

Lo primero que se observa es cómo la máquina ha guardado en memoria el valor `0x110` (si aplicamos ceros a la izquierda tenemos el valor `0x00000110`). Se observa la aplicación de la norma "little-endian" para el almacenamiento de datos.

El primer byte indica el tipo de operación "mov" y cuál es el registro destino "edx". Los cuatro últimos bytes indican el valor que tiene que asignarse al registro seleccionado. Pasando el primer byte a binario se observa lo siguiente:

```
1011 1010
```

Los bits 4, 5 , 6 y 7 (dato `1011`) indican que se trata de la operación "mov", el bit 3 indica que se trata de mover un dato de tamaño "double word" y los bits 0, 1 y 2 (dato `010`) indican que los datos se tienen que guardar en el registro "edx".

El registro destino estaría especificado por la siguiente relación:

- `000` → `eax`
- `001` → `ecx`
- `010` → `edx`
- `011` → `ebx`
- `100` → `esp`

- 101 → ebp
- 110 → esi
- 111 → edi

Aún así, esta relación depende del tipo de instrucción que se esté ejecutando. Si en lugar de ser una instrucción "mov" de tamaño "double word" hubiera sido una instrucción de tamaño de byte, la relación que definiría el registro destino sería otra:

- 000 → al
- 001 → cl
- 010 → dl
- 011 → bl
- 100 → ah
- 101 → ch
- 110 → dh
- 111 → bh

Si se quisiera hacer un cambio en el tamaño de los operandos y pasar a una asignación de byte, la sentencia en ensamblador, aunque incorrecta, sería la siguiente:

```
mov $0x110, %dl
```

En esta sentencia se está intentando asignar un valor que supera el valor del tamaño máximo de un byte. El valor `0x110` es mayor a un byte, así que si se quiere que la sentencia sea correcta, se debe cambiar el valor o cambiar el registro destino.

```
mov $0x10, %dl
```

Esta sentencia sí es correcta y su traducción a código máquina en hexadecimal es:

```
b2 10
```

Se observa que el código generado es menor que en la instrucción anterior, puesto que el tamaño de los operandos ha variado. Esto tiene fuertes implicaciones si se modifica código máquina en ejecución, ya que si se hace esta modificación de la primera sentencia que se ha estudiado:

```
ba 10 01 00 00      mov $0x110, %edx
b2 10               mov $0x10, %dl
```

se aprecia que lo único que ha cambiado es el código de instrucción (ha pasado de `baab2`). De esta manera, el tamaño de los operandos ha cambiado y tres de los bytes que antes formaban parte del valor a asignar al registro destino quedan ahora sin función.

Estos bytes no quedan sin hacer nada durante la ejecución del programa, sino que el sistema los interpreta como si fueran una nueva instrucción. La norma que sigue el sistema es: "donde termina una instrucción, empieza otra". De esta manera, unos bytes que debían ser interpretados como tales pasan a ser código ejecutable, alterando así la ejecución normal del programa. En el ejemplo, los dos siguientes bytes serían ejecutados como:

```
01 00 add %eax,(%eax)
```

y a partir de este punto se ejecutarían un sinfín de cambios debido a la reinterpretación del código máquina por el sistema.

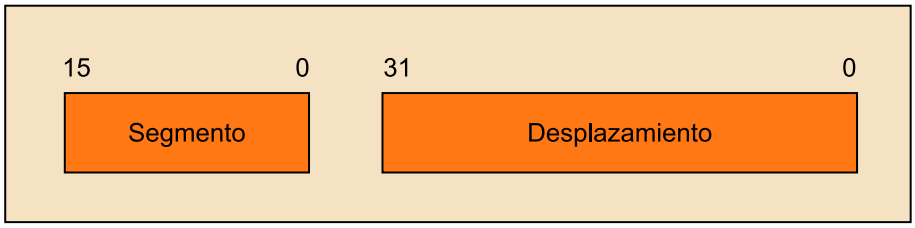
3.3. Direccionamiento y operandos

Como se ha visto, las instrucciones en ensamblador requieren operandos. El número de operandos puede variar de 0 (operandos implícitos a la instrucción) a varios (operandos explícitos). Los datos que proveen los operandos pueden venir especificados por:

- **Un valor explícito expresado en la instrucción (valor inmediato)**. Los valores inmediatos tan sólo pueden ser usados como operandos fuente de una instrucción, puesto que están escritos directamente en la instrucción y, por lo tanto, no expresan ningún contenedor válido para registrar datos. Son, solamente, la expresión de un valor.
  Muchas operaciones, sobre todo las aritméticas, aceptan valores inmediatos como operandos fuente. El valor máximo expresado por estos operandos y aceptado por las distintas instrucciones varía de instrucción a instrucción.

- **Un registro**. Los registros pueden ser especificados tanto como operandos fuente como operandos destino. Todos los registros de propósito general vistos pueden ser usados (los de 32, 16 y 8 bits). Los registros de segmento también. El registro EFLAGS es usado de forma implícita en algunas instrucciones (como por ejemplo, en las instrucciones de salto condicional).

- **Una dirección de memoria**. Un operando especificando una dirección de memoria siempre implica el uso de un segmento y un desplazamiento dentro del segmento.



Acceso a memoria

La selección del segmento a utilizar para el acceso a memoria se puede hacer de manera implícita o explícita. Lo más habitual es cargar los diferentes registros de segmento con los valores correspondientes y dejar que la instrucción ejecutada seleccione el segmento por defecto. El procesador sigue una serie de reglas para escoger los registros de segmento durante la ejecución de una instrucción:

- El registro de segmento CS se usa siempre que hay que acceder a una instrucción nueva para cargarla al registro EIP.

- El registro de segmento SS se usa para los accesos a la pila. Todas las instrucciones que utilicen los registros EBP o ESP para acceder a datos de la pila usarán el registro SS como registro de segmento base.

- El registro de segmento DS se usa cuando se accede a referencias a datos, excepto aquellas que se ubiquen en la pila, o hagan referencia al destino de cadenas de caracteres *(strings)*.

- El registro de segmento ES para ubicar el destino de las instrucciones que hagan operaciones sobre cadenas de caracteres.

Aunque los segmentos se usan muchas veces de manera implícita, también se pueden usar de manera explícita, expresando el segmento que se quiere usar para alterar así el orden preestablecido por el sistema.

```
mov $0x15, %es:%ebx
```

Con la instrucción anterior se cargaría el valor 15 en hexadecimal, en la dirección de memoria expresada por la combinación del registro ES (segmento) y EBX (desplazamiento).

Aunque es posible hacer este tipo de llamadas, hay algunas normas que no pueden ser alteradas.

- Las instrucciones siempre se cargarán usando el registro CS.
- El destino de las operaciones de *strings* siempre será ubicado en el segmento ES.
- Las instrucciones de *push* y *pop* siempre se ejecutan usando el registro SS.

El acceso a memoria es un punto clave en la arquitectura de computadores. Para acceder a ella existen varios métodos de direccionamiento en los cuales los desplazamientos se expresan de diferentes formas para poder acceder a los datos. Este desplazamiento se puede expresar como un valor estático o como el resultado de uno o varios componentes:

- Puntero base.
- Índice.

- Escala.
- Desplazamiento.

Estos cuatro elementos se suelen combinar mediante la siguiente fórmula:

puntero_base + índice * escala + desplazamiento

Éste cálculo es especialmente útil para acceder a matrices de datos guardados en memoria. No hay que usar todos los elementos en cada acceso a memoria. A continuación se exponen algunos ejemplos de acceso a memoria en la sintaxis de Intel y la de AT&T:

- Accediendo a los datos apuntados por la dirección de memoria contenida en un registro:

AT&T:	`(%eax)`
Intel:	`[eax]`

- Añadiendo un desplazamiento:

AT&T:	`_despl(%eax)`
Intel:	`[eax + _despl]`

- Accediendo a un valor en una matriz de enteros de tamaño 4 bytes:

AT&T:	`_array(,%eax,4)`
Intel:	`[eax*4 + _array]`

- Accediendo a un carácter en una matriz de registros de 8 caracteres, en donde eax contiene el número de registros deseado y ebx tiene el desplazamiento deseado dentro del registro:

AT&T:	`_array(%ebx, %eax, 8)`
Intel:	`[ebx + eax*8 + _array]`

A nivel de código máquina las distintas combinaciones de desplazamiento, base, escala e índice, se codifican en una instrucción. Los compiladores de lenguajes de alto nivel seleccionan los elementos más apropiados para acceder a memoria teniendo en cuenta las estructuras creadas por los programadores.

- **Un puerto de entrada/salida**. El espacio de entrada/salida es accedido mediante un conjunto de puertos. El número de puertos disponible es li-

mitado. Estos son accesibles mediante un operando con un valor inmediato o bien un valor en el registro DX.

3.4. Desensamblando un pequeño programa

Vista la pequeña introducción a lenguaje ensamblador, se considera el siguiente programa en lenguaje C.

```
#include <stdio.h>
int main()
{

   int x, y, result;

   x = 3;

   y = 4;

   result = x + y;

   printf("Resultado de x + y = %d\n", result);

   return(1);
}
```

El programa tan sólo realiza la suma de dos variables locales (`x` e `y`) declaradas de manera local en el propio cuerpo principal del programa. El resultado es guardado en la variable `result` para ser posteriormente volcado por pantalla mediante la función `printf`.

Se procede a compilar el programa y posteriormente a desensamblar, para ver cuál es el resultado a niveles de programación más bajos: código máquina y ensamblador. Se realiza la operación en dos entorno completamente distintos: Linux + GCC[5] y Windows + BCC[6].

(5)**GCC, the GNU Compiler Collection**. Disponible en: http://gcc.gnu.org/

(6)**Borland C++ compiler**. Disonible en: http://www.codegear.com/downloads/free/cppbuilder

3.4.1. Entorno: Linux + GCC

El código resultante de desensamblar el ejecutable, resultado de compilar el programa anterior centrando el objetivo en el código escrito en la función main, es:

```
Columna 1                        Columna 2     Columna 3

080483a4 <main>:
080483a4: 8d 4c 24 04               lea      0x4(%esp),%ecx
080483a8: 83 e4 f0                  and      $0xfffffff0,%esp
080483ab: ff 71 fc                  pushl    -0x4(%ecx)
080483ae: 55                        push     %ebp
080483af: 89                        e5 mov   %esp,%ebp
080483b1: 51                        push     %ecx
080483b2: 83 ec 24                  sub      $0x24,%esp
080483b5: c7 45 f0 03 00 00 00      movl     $0x3,-0x10(%ebp)
080483bc: c7 45 f4 04 00 00 00      movl     $0x4,-0xc(%ebp)
080483c3: 8b 55 f4                  mov      -0xc(%ebp),%edx
080483c6: 8b 45 f0                  mov      -0x10(%ebp),%eax
080483c9: 01 d0                     add      %edx,%eax
080483cb: 89 45 f8                  mov      %eax,-0x8(%ebp)
080483ce: 8b 45 f8                  mov      -0x8(%ebp),%eax
080483d1: 89 44 24 04               mov      %eax,0x4(%esp)
080483d5: c7 04 24 b0 84 04 08      movl     $0x80484b0,(%esp)
080483dc: e8 f7 fe ff ff            call     80482d8 <printf@plt>
080483e1: b8 01 00 00 00            mov      $0x1,%eax
080483e6: 83 c4 24                  add      $0x24,%esp
080483e9: 59                        pop      %ecx
080483ea: 5d                        pop      %ebp
080483eb: 8d 61 fc                  lea      -0x4(%ecx),%esp
080483ee: c3                        ret
080483ef: 90                        nop
```

La sintaxis AT&T está muy extendida en el mundo UNIX, así que la mayoría de utilidades que trabajan bajo estas plataformas suele usarla, aunque existen herramientas que también permiten el desensamblaje en la sintaxis de Intel.

En el listado resultante hay pocas cosas identificables a primera vista. Las distintas columnas indican lo siguiente:

- Primera columna. Dirección de memoria donde se encuentra la instrucción. Estas son las direcciones de memoria que se irán cargando en el registro EIP para seguir la ejecución, comando a comando, del programa.

- Segunda columna. Se muestran los códigos en lenguaje máquina *(opcodes)* de cada instrucción, resultado de la compilación del programa.

- Tercera columna. Instrucciones en lenguaje ensamblador resultado de desensamblar los códigos en lenguaje máquina de la segunda columna.

Así pues, en cada línea se observa la dirección de memoria de la instrucción, los bytes correspondientes al código máquina y la instrucción correspondiente al código máquina en lenguaje ensamblador.

Si se observa la línea:

```
080483dc: e8 f7 fe ff ff   call 80482d8 <printf@plt>
```

se puede ver cómo el sistema realiza una llamada a una función llamada `printf`. Se produce, en este punto, un salto en la ejecución lineal del programa a la dirección `080482d8`, que es en donde se encuentra ubicada la función `printf`.

El código máquina indica la función `call` mediante el *opcode* `E8`. Los 4 bytes restantes indican el desplazamiento. Si se tiene en cuenta que los datos se almacenan siguiendo la norma "little endian" se ve que el desplazamiento indicado es `fffffef7`, que pasado a decimal corresponde con el número -265.

Si se realiza la resta entre la dirección destino y la dirección a la instrucción siguiente a la ejecución del call, se obtiene que:

```
Hexadecimal: 80482d8 - 80483e1 = fffffef7
Decimal: 134513368 - 134513633 = -265
```

Obtenemos así el desplazamiento indicado en el código máquina. Esto indica que la llamada va a ser un `call near` a una dirección dentro del mismo segmento de código. Es importante recordar esto, ya que los *exploits* modificaran la dirección de retorno.

A continuación se intercala el código fuente escrito en C con el código en ensamblador: (más adelante hablaremos de una herramienta que nos permitirá ver este código ensamblador a partir de código C)

```
Dirección                       Opcodes         Assembler

080483a4 <main>:
#include <stdio.h>

int main()
{
80483a4: 8d 4c 24 04              lea         0x4(%esp),%ecx
80483a8: 83 e4 f0                 and         $0xfffffff0,%esp
80483ab: ff 71 fc                 pushl       -0x4(%ecx)
80483ae: 55                       push        %ebp
80483af: 89 e5                    mov         %esp,%ebp
80483b1: 51                       push        %ecx
80483b2: 83 ec 24                 sub         $0x24,%esp
int x, y, result;

x = 3;
80483b5: c7 45 f0 03 00 00 00     movl        $0x3,-0x10(%ebp)
y = 4;
80483bc: c7 45 f4 04 00 00 00     movl        $0x4,-0xc(%ebp)

result = x + y;
80483c3: 8b 55 f4                 mov         -0xc(%ebp),%edx
80483c6: 8b 45 f0                 mov         -0x10(%ebp),%eax
80483c9: 01 d0                    add         %edx,%eax
80483cb: 89 45 f8                 mov         %eax,-0x8(%ebp)

printf("Resultado de x + y = %d\n", result);
80483ce: 8b 45 f8                 mov         -0x8(%ebp),%eax
80483d1: 89 44 24 04              mov         %eax,0x4(%esp)
80483d5: c7 04 24 b0 84 04 08     movl        $0x80484b0,(%esp)
80483dc: e8 f7 fe ff ff           call        80482d8 <printf@plt>

return(1);
80483e1: b8 01 00 00 00           mov         $0x1,%eax
80483e6: 83 c4 24                 add         $0x24,%esp
80483e9: 59                       pop         %ecx
80483ea: 5d                       pop         %ebp
80483eb: 8d 61 fc                 lea         -0x4(%ecx),%esp
80483ee: c3                       ret
}
```

Se pueden ver las líneas de código en C intercaladas en medio de las líneas de código en ensamblador. Las líneas en ensamblador que aparecen después de cada línea de código en C son las necesarias para que la instrucción en C sea ejecutada. Hay una excepción a esta norma y es la declaración de variables a la entrada de la función `main`. La declaración de los enteros aparece sin líneas en ensamblador a ejecutarse. Esto no significa que esa línea no tenga repercusión a nivel ensamblador, sino que las operaciones necesarias para la ejecución de esa instrucción ya se han ejecutado previamente.

```
int x, y, result;
```

Al ser variables locales, la traducción de la declaración de las variables sería la reserva de espacio en la pila para estas últimas. Esta reserva de espacio se hace en la línea inmediatamente anterior a la declaración de las variables:

```
080483b2: 83 ec 24    sub $0x24,%esp
```

Esta instrucción reserva 36 bytes en la pila para ser utilizados de manera local. Entre los usos se encuentra la utilización de las variables locales.

3.4.2. Entorno: Windows + BorlandC (BCC)

El código resultante de desensamblar el ejecutable, resultado de compilar el programa anterior centrando el objetivo en el código escrito en la función `main`, es:

```
Columna 1      Columna 2      Columna 3

00401150 >/.   55             PUSH EBP
00401151 |.    8BEC           MOV EBP,ESP
00401153 |.    B8 03000000    MOV EAX,3
00401158 |.    BA 04000000    MOV EDX,4
0040115D |.    03D0           ADD EDX,EAX
0040115F |.    8BC2           MOV EAX,EDX
00401161 |.    50             PUSH EAX
00401162 |.    68 28A14000    PUSH EJEMPLO1.0040A128
00401167 |.    E8 10270000    CALL EJEMPLO1.0040387C
0040116C |.    83C4 08        ADD ESP,8
0040116F |.    B8 01000000    MOV EAX,1
00401174 |.    5D             POP EBP
00401175 \.    C3             RETN
```

La sintaxis Intel es la más utilizada en sistemas basados en Windows, así que la mayoría de utilidades que trabajan bajo los sistemas de Microsoft suele usarla.

En el listado resultante hay pocas cosas identificables a primera vista. Las distintas columnas indican lo siguiente:

- Primera columna. Dirección de memoria donde se encuentra la instrucción. Esta es la dirección que se cargará en el registro EIP para seguir la ejecución, comando a comando, del programa.

- Segunda columna. Se muestran los códigos en lenguaje máquina *(opcodes)* de cada instrucción, resultado de la compilación del programa.

- Tercera columna. Instrucciones en lenguaje ensamblador resultado de desensamblar los códigos en lenguaje máquina de la segunda columna.

Así pues, en cada línea se observa la dirección de memoria de la instrucción, los bytes correspondientes al código máquina y la instrucción correspondiente al código máquina en lenguaje ensamblador.

Se observan, por lo tanto, los mismos datos que en el caso de "linux+gcc". Para continuar la analogía con el caso anterior, se observa la línea que ejecuta la instrucción `call`:

```
00401167 |. E8 10270000  CALL EJEMPLO1.0040387C
```

Se puede ver cómo el sistema realiza una llamada a una función, y en este caso no sabemos si se trata de la función `printf`, a menos que tengamos los códigos de las funciones de Windows. Se produce en este punto un salto en la ejecución lineal del programa a la dirección `0040387C`, que es en donde se debe encontrar la entrada a la llamada a la función `printf`.

El código máquina indica la función `call` mediante el *opcode* `E8`, igual que en el caso anterior. Los 4 bytes restantes indican el desplazamiento. Si se tiene en cuenta que los datos se almacenan siguiendo la norma "little endian" se ve que el desplazamiento indicado es `00002710`, que pasado a decimal se corresponde con el número `10000`. Si se realiza la resta entre la dirección destino y la dirección a la instrucción siguiente a la ejecución del `call`, se obtiene que:

```
Hexadecimal: 0040387C - 0040116C = 2710
Decimal: 4208764 - 4198764 = 10000
```

Obtenemos el desplazamiento indicado en el código máquina. Esto quiere decir que la llamada va a ser un `call near` a una dirección dentro del mismo segmento de código.

A continuación se intercala el código fuente escrito en C con el código en ensamblador:

```
Dirección            Opcodes          Assembler
int main()
{
   int x, y, result;
   00401150 >/.      55               PUSH EBP
   00401151 |.       8BEC             MOV EBP,ESP

   x = 3;
   00401153 |.       B8 03000000      MOV EAX,3

   y = 4;
   00401158 |.       BA 04000000      MOV EDX,4

   result = x + y;
   0040115D |.       03D0             ADD EDX,EAX

   printf("Resultado de x + y = %d\n", result);
   0040115F |.       8BC2             MOV EAX,EDX
   00401161 |.       50               PUSH EAX
   00401162 |.       68 28A14000      PUSH EJEMPLO1.0040A128
   00401167 |.       E8 10270000      CALL EJEMPLO1.0040387C
   0040116C |.       83C4 08          ADD ESP,8

   return(1);
   0040116F |.       B8 01000000      MOV EAX,1
   00401174 |.       5D               POP EBP
   00401175 \.       C3               RETN
}
```

Se observa que las variables locales definidas no reservan espacio en la pila, sino que se utilizan directamente registros del procesador para realizar la tarea de las variables locales.

3.4.3. Comparativa de los entornos

Aparte del hecho de que los dos entornos suelen utilizar dos sintaxis distintas a la hora de desensamblar los programas, existen otras diferencias. La primera, y más importante, es que con compiladores distintos tenemos resultados distintos en código máquina,. Así pues, se ha visto que en el ejemplo de "linux + gcc" se reservaba espacio en la pila para las variables locales, y en el ejemplo de "windows + bcc" esta reserva no se realizaba y se utilizaban los registros a modo de variables locales.

Es tarea del compilador crear el código máquina de tal manera que los resultados esperados detallados por el programa de alto nivel se consigan. El programador no tiene control sobre cómo se crea el código máquina final, tan sólo puede confiar en la garantía que le ofrece el compilador en sí. Así pues, si se dispone de compiladores distintos, el código máquina resultante será, seguramente, distinto para cada uno de los casos. Se coincide entonces con una norma de la programación que indica que un mismo programa (mismas entradas y mismas salidas) puede ser escrito de infinitas maneras diferentes.

Se comparan ambos programas:

Windows + BCC	**Linux + GCC**
Opcodes Assembler	**Opcodes Assembler**
int main()	**int main()**
{	**{**
	8d 4c 24 04 lea 0x4(%esp),%ecx
	83 e4 f0 and $0xfffffff0,%esp
	ff 71 fc pushl -0x4(%ecx)
55 PUSH EBP	55 push %ebp
8BEC MOV EBP,ESP	89 e5 mov %esp,%ebp
	51 push %ecx
	83 ec 24 sub $0x24,%esp
int x, y, result;	**int x, y, result;**
x = 3;	**x = 3;**
B8 03000000 MOV EAX,3	c7 45 f0 03000000 movl $0x3,-x10(%ebp)
y = 4;	**y = 4;**
BA 04000000 MOV EDX,4	c7 45 f4 04000000 movl $0x4,-xc(%ebp)

result = x + y;	**result = x + y;**
03D0 ADD EDX,EAX	8b 55 f4 mov -0xc(%ebp),%edx
	8b 45 f0 mov -0x10(%ebp),%eax
	01 d0 add %edx,%eax
	89 45 f8 mov %eax,-0x8(%ebp)
printf("Resultado de	**printf("Resultado de**
x + y = %d\n", result);	**x + y = %d\n", result);**
8BC2 MOV EAX,EDX	8b 45 f8 mov -0x8(%ebp),%eax
50 PUSH EAX	89 44 24 04 mov %eax,0x4(%esp)
68 28A14000 PUSH EJEMPLO1.0040ª128	c7 04 24 b0840408 movl $0x80484b0, (%esp)
E8 10270000 CALL EJEMPLO1.0040387C	e8 f7feffff call 80482d8 <printf@plt>
83C4 08 ADD ESP,8	
return(1);	**return(1);**
B8 01000000 MOV EAX,1	b8 01000000 mov $0x1,%eax
	83 c4 24 add $0x24,%esp
	59 pop %ecx
5D POP EBP	5d pop %ebp
	8d 61 fc lea -0x4(%ecx),%esp
C3 RETN	c3 ret
}	**}**

- Entrada a la función principal. En "Windows+BCC" la entrada a la función principal es claramente más simple. No hay reserva de espacio en la pila para las variables locales y no se hacen una serie de operaciones como ocurre en el otro caso (operaciones internas de control en el caso "Linux+gcc"). Esto no tiene ninguna implicación en cuanto a la ejecución, tan sólo se remarca el hecho de que el resultado de compilar en los dos entornos es diferente.

- Asignaciones a las variables locales (`x=3`, `y=4`). En el caso "Windows+BCC" se asignan los distintos valores directamente a registros; en concreto, los registros EAX y EDX. En el caso "Linux+gcc" los valores se asignan al espacio que tienen las variables locales en la pila. Esto provoca que el código máquina resultante en el caso "Windows+BCC" sea más reducido.

- Cálculo del resultado final y asignación del resultado a una variable local (`result = x + y`). En el caso "Windows+BCC" se realiza con la instrucción `add` aplicada directamente contra los dos registros implicados y guardando el resultado en uno de ellos (eax). En el otro caso, primero se cargan los valores de las variables locales a registros, después se realiza la suma y finalmente se guarda el valor final en el espacio asignado a las variables locales en la pila.

- Llamada a la función `printf`. En ambos casos se introducen los parámetros necesarios en la pila para poder hacer la llamada con `call` de manera adecuada. En el caso de "Windows+BCC" se empilan los elementos mediante la instrucción `push`, y en el caso "Linux+gcc" se empilan los elementos asignando los valores adecuados dentro de un espacio de la pila reservado previamente para realizar llamadas a funciones. Al finalizar la llamada (`call`) no se quitan los parámetros introducidos en la pila, ya que ese espacio está reservado para ese propósito. En "Windows+BCC" se hace necesario quitar los elementos previamente introducidos, incrementando el valor de "esp".
  Como detalle a comentar, cabe destacar que la llamada en "Windows+BCC" se realiza hacia algún punto ubicado más hacia adelante del punto actual y el el caso "Linux+gcc" la llamada es hacia a algún punto anterior.

- Salida de la función principal del programa (`return`). En el caso "Linux+gcc" se realizan algunas operaciones internas más que en el otro caso. Aun así, ambos sistemas coinciden en asignar el valor 1 al registro "eax", recuperar el valor anterior de "ebp" de la pila, y ejecutar la instrucción "ret". En "Windows+BCC" la última instrucción es `retn`, y en "Linux+gcc" la última instrucción es "ret". Parece que se trate de dos instrucciones distintas, aunque si se observa el *opcode* correspondiente a la instrucción en ambos casos, se ve que los dos tienen asignado el *opcode* "c3". Se ejecuta, por lo tanto, la misma instrucción en ambos casos.